A Framework for Point-free Program Transformation*

Alcino Cunha, Jorge Sousa Pinto, and José Proença

CCTC / Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
{alcino,jsp,jproenca}@di.uminho.pt

Abstract. The subject of this paper is functional program transformation in the so-called *point-free* style. By this we mean first translating programs to a form consisting only of categorically-inspired combinators, algebraic data types defined as fixed points of functors, and implicit recursion through the use of type-parameterized recursion patterns. This form is appropriate for reasoning about programs equationally, but difficult to actually use in practice for programming. In this paper we present a collection of libraries and tools developed at Minho with the aim of supporting the automatic conversion of programs to *point-free* (embedded in Haskell), their manipulation and rule-driven simplification, and the (limited) automatic application of *fusion* for program transformation.

1 Introduction

Functional Programming has always been known to be appropriate for activities involving manipulation of programs, such as program transformation. This is due to the strong theoretical basis that underlies the programming languages: the semantics of functional programs are easier to formalize.

As with any programming paradigm, different functional programmers use different styles of programming; it is however true that most advanced programmers resort to some concise form where functions are written as combinations of other functions, rather than programming by explicit manipulation of the arguments and explicit recursion. For instance a function that sums the squares of the elements in a list can be written in Haskell as

```
sum_squares = (foldr (+) 0) . (map sq)  where sq x = x*x
```

A radical style of programming is the so-called *point-free* style, which totally dispenses with variables. For instance the function `sq` above can be written as `sq = mult . (id /\ id)`, where the infix operator `/\` corresponds to the *split* combinator that applies two functions to an argument, producing a pair, and `mult` is the uncurried product.

The origins of the point-free style can be traced back to the ACM Turing Award Lecture given by John Backus in 1977 [1]. Instead of explicitly referring

* This work was partially supported by FCT project PURe (POSI/CHS/44304/2002).

arguments, Backus recommended the use of *functional forms* (combinators) to build functions by combining simpler ones. The particular choice of combinators should be driven by the associated algebraic laws.

In the modern incarnation of these ideas, the combinators correspond to morphisms in a category (where the denotational semantics of the language are constructed) and the desired laws follow directly from universal properties of this category. What is more, this approach extends smoothly to the treatment of recursion in what is known as the *data type-generic* approach to programming [8, 14]. This allows one to reason equationally about functions obtained by applying standard *recursion patterns*, thus replacing the use of fixpoint induction.

The generic aspect of this approach comes from the fact that all the constructions are parameterized by the recursive data types involved in the computations. It is widely accepted that this style is a good choice for reasoning about programs equationally and generically. It has also proved to be fruitful in the field of *program transformation* [4], where well-known concepts like folding or fusion over lists were first introduced by Bird to derive accumulator-based implementations from inefficient specifications [2].

As a simple example of the kind of transformation we mean, in the function `sum_squares` given above, the fold can be equationally fused with the map to give the following one-pass function, where `plus` is uncurried sum.

```
sum_squares' = foldr aux 0  where aux = curry (plus . (sq . fst /\ snd))
```

The drawback of using this radical point-free style is that, as the examples in this paper show, programs written without variables are not always easy to write or understand. In fact, it is virtually impossible to program without using variables here and there. Pointwise vs. point-free is a lively discussion subject in Haskell forums; the goal of the present paper is to present a set of libraries and tools that support point-free program transformation, but this includes the automatic translation of code to point-free form, so that programmers may apply point-free techniques to their code with variables.

Specifically, we present here the following components, which are all freely available as part of the **UMinho Haskell Software** distribution.

Pointless: a library for point-free programming, allowing programmers to type-check and execute point-free code with recursion patterns, parameterized by data types. With the help of extensions to the Haskell type system, we have implemented an implicit coercion mechanism that provides a limited form of structural equivalence between types. This has allowed us to embed in Haskell a syntax almost identical to the one used at the theoretical level.

DrHylo: a tool that allows programmers to automatically convert Haskell code to point-free form with recursion patterns. In particular, we employ the well-known equivalence between simply typed λ-calculi and cartesian closed categories suggested by Lambek [12]. This serves as the basis for the translation of a core functional language to categorical combinators, extended by the first author [3] to cover sum types. A second component here is the application

$$\frac{}{\Gamma \vdash \star : 1} \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma[x \mapsto A] \vdash M : B}{\Gamma \vdash \lambda x.\ M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \qquad \frac{\Gamma \vdash L : A + B \quad \Gamma \vdash M : A \to C \quad \Gamma \vdash N : B \to C}{\Gamma \vdash \mathsf{case}\ L\ M\ N : C}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\ M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\ M : B} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{inl}\ M : A + B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathsf{inr}\ M : A + B}$$

$$\frac{\Gamma \vdash M : F\ (\mu F)}{\Gamma \vdash \mathsf{in}_{\mu F}\ M : \mu F} \qquad \frac{\Gamma \vdash M : \mu F}{\Gamma \vdash \mathsf{out}_{\mu F}\ M : F\ (\mu F)} \qquad \frac{\Gamma \vdash M : A \to A}{\Gamma \vdash \mathsf{fix}\ M : A}$$

Fig. 1. Typing rules

of a standard algorithm that converts recursive functions to *hylomorphisms* of adequate regular data-types, thus removing explicit recursion.

SimpliFree: a tool for manipulating point-free code. Its use will be exemplified here: (i) for the simplification of the very verbose terms produced by DrHylo; and (ii) for program transformation by applying fold fusion.

Organization of the Paper. Section 2 introduces the languages used in the paper, and Sect. 3 reviews notions of point-free equational reasoning, with the help of an example. Pointless, DrHylo, and SimpliFree are described in Sects. 4, 5 and 6. Finally Sect. 7 concludes the paper.

2 The Pointwise and Point-free Styles of Programming

In both styles, types are defined according to the following syntax.

$$A, B ::= 1 \mid A \to B \mid A \times B \mid A + B \mid \mu F$$
$$F, G ::= \mathsf{Id} \mid \underline{A} \mid F \otimes G \mid F \oplus G \mid F \circledcirc G$$

We assume a standard domain-theoretic semantics, where types are pointed complete partial orders, with least element $\bot$. 1 is the single element type, $A \to B$ is the type of continuous functions from A to B, $A \times B$ is the cartesian product, $A + B$ is the separated sum (with distinguished least element), and μF is a recursive (regular) type defined as the fixed point of functor F.

Id denotes the identity functor, $\underline{A}$ the constant functor that always returns A, $\otimes$ and $\oplus$ the lifted product and sum bifunctors, and $\circledcirc$ composition of functors. For example, booleans can be defined as $\mathsf{Bool} = 1 + 1$, natural numbers as $\mathsf{Nat} = \mu(\underline{1} \oplus \mathsf{Id})$, and lists with elements of type A as $\mathsf{List}\ A = \mu(\underline{1} \oplus \underline{A} \otimes \mathsf{Id})$.

Pointwise Language. Terms with variables are generated by the grammar

$$L, M, N ::= \star \mid x \mid M\ N \mid \lambda x.\ M \mid \langle M, N \rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid$$
$$\mathsf{case}\ L\ M\ N \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid \mathsf{in}_{\mu F}\ M \mid \mathsf{out}_{\mu F}\ M \mid \mathsf{fix}\ M$$

Apart from variable, abstraction, and application, we find $\star$, which is the unique inhabitant of the terminal type (as such, it equals $\bot_1$); fst and snd are projections

from a product type and inl and inr are injections into a sum type; $\langle \cdot, \cdot \rangle$ is a pairing construct, and case performs case-analysis on sums. Associated with each recursive type μF are two unique strict functions $\mathsf{in}_{\mu F}$ and $\mathsf{out}_{\mu F}$, that are each other's inverse. These provide the means to construct and inspect values of the given type. Whenever clear from context, the subscripts will be omitted.

The typing rules are presented in Fig. 1. We now show examples of terms in this language.

$$
\begin{array}{ll}
\mathsf{zero} \;:\; \mathsf{Nat} & \mathsf{nil} \;:\; \mathsf{List}\ A \\
\mathsf{zero} = \mathsf{in}\ (\mathsf{inl}\ \star) & \mathsf{nil} = \mathsf{in}\ (\mathsf{inl}\ \star) \\
\mathsf{succ} \;:\; \mathsf{Nat} \to \mathsf{Nat} & \mathsf{cons} \;:\; A \to \mathsf{List}\ A \to \mathsf{List}\ A \\
\mathsf{succ} = \lambda x.\ \mathsf{in}\ (\mathsf{inr}\ x) & \mathsf{cons} = \lambda ht.\ \mathsf{in}\ (\mathsf{inr}\ \langle h, t \rangle) \\
\mathsf{swap} \;:\; A \times B \to B \times A & \mathsf{null} \;:\; \mathsf{List}\ A \to \mathsf{Bool} \\
\mathsf{swap} = \lambda x.\ \langle \mathsf{snd}\ x, \mathsf{fst}\ x \rangle & \mathsf{null} = \lambda l.\mathsf{case}\ (\mathsf{out}\ l)\ (\lambda x.\mathsf{true})\ (\lambda x.\mathsf{false}) \\
\multicolumn{2}{l}{\mathsf{distr} \;:\; A \times (B + C) \to (A \times B) + (A \times C)} \\
\multicolumn{2}{l}{\mathsf{distr} = \lambda x.\ \mathsf{case}\ (\mathsf{snd}\ x)\ (\lambda y.\ \mathsf{inl}\ \langle \mathsf{fst}\ x, y \rangle)\ (\lambda y.\ \mathsf{inr}\ \langle \mathsf{fst}\ x, y \rangle)}
\end{array}
$$

Recursive functions are defined explicitly using fix. For example, assuming that $\mathsf{mult} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$, the factorial and length functions can be defined as

$$
\begin{array}{l}
\mathsf{fact} \;:\; \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{fact} = \mathsf{fix}\ (\lambda f.\ \lambda x.\ \mathsf{case}\ (\mathsf{out}\ x)\ (\lambda y.\ \mathsf{succ}\ \mathsf{zero})\ (\lambda y.\ \mathsf{mult}\ \langle \mathsf{succ}\ y, f\ y \rangle)) \\
\mathsf{length} \;:\; \mathsf{List}\ A \to \mathsf{Nat} \\
\mathsf{length} = \mathsf{fix}\ (\lambda f.\ \lambda l.\ \mathsf{case}\ (\mathsf{out}\ l)\ (\lambda x.\ \mathsf{zero})\ (\lambda x.\ \mathsf{succ}\ (f\ (\mathsf{snd}\ y))))
\end{array}
$$

Point-free Language. The set of combinators that is of interest to us comes from universal constructions in *almost bicartesian closed categories*, that is, categories with products, non-empty sums, exponentials, and terminal object. See for instance [13] for a thorough treatment of the subject.

The point-free language contains the constants fst, snd, inl, inr, in, and out, with the obvious types, and also the set of combinators given below. To convey the meaning of each combinator, we give its definition in the pointwise language.

$$
\begin{array}{ll}
(\cdot \circ \cdot) \;:\; (B \to C) \to (A \to B) \to A \to C & \mathsf{id} \;:\; A \to A \\
(\cdot \circ \cdot) = \lambda fgx.\ f\ (g\ x) & \mathsf{id} = \lambda x.\ x \\
(\cdot \vartriangle \cdot) \;:\; (A \to B) \to (A \to C) \to A \to (B \times C) & \mathsf{bang} \;:\; A \to 1 \\
(\cdot \vartriangle \cdot) = \lambda fgx.\ \langle f\ x, g\ x \rangle & \mathsf{bang} = \lambda x.\ \star \\
(\cdot \triangledown \cdot) \;:\; (A \to C) \to (B \to C) \to (A + B) \to C & \mathsf{ap} \;:\; (A \to B) \times A \to B \\
(\cdot \triangledown \cdot) = \lambda fgx.\ \mathsf{case}\ x\ (\lambda y.\ f\ y)\ (\lambda y.\ g\ y) & \mathsf{ap} = \lambda x.\ (\mathsf{fst}\ x)\ (\mathsf{snd}\ x) \\
\overline{\cdot} \;:\; (A \times B \to C) \to A \to B \to C & \\
\overline{\cdot} = \lambda fxy.\ f\ \langle x, y \rangle &
\end{array}
$$

It is convenient to have derived combinators corresponding to the operation of the product, sum, and exponentiation functors. These can be defined, respectively, as $f \times g = f \circ \mathsf{fst} \vartriangle g \circ \mathsf{snd}$, $f + g = \mathsf{inl} \circ f \triangledown \mathsf{inr} \circ g$, and $f^{\bullet} = \overline{f \circ \mathsf{ap}}$.

The point-free language contains only values of functional type. As such, elements of a non-functional type A are denoted by functions of the isomorphic

type $1 \to A$. The previous examples can be written in the point-free language:

$$\begin{array}{ll}
\mathsf{zero} \;:\; 1 \to \mathsf{Nat} & \mathsf{nil} \;:\; 1 \to \mathsf{List}\ A \\
\mathsf{zero} = \mathsf{in} \circ \mathsf{inl} & \mathsf{nil} = \mathsf{in} \circ \mathsf{inl} \\
\mathsf{succ} \;:\; \mathsf{Nat} \to \mathsf{Nat} & \mathsf{cons} \;:\; A \to \mathsf{List}\ A \to \mathsf{List}\ A \\
\mathsf{succ} = \mathsf{in} \circ \mathsf{inr} & \mathsf{cons} = \overline{\mathsf{in} \circ \mathsf{inr}} \\
\mathsf{swap} \;:\; A \times B \to B \times A & \mathsf{null} \;:\; \mathsf{List}\ A \to \mathsf{Bool} \\
\mathsf{swap} = \mathsf{snd} \vartriangle \mathsf{fst} & \mathsf{null} = (\mathsf{true} \triangledown \mathsf{false} \circ \mathsf{bang}) \circ \mathsf{out} \\
\multicolumn{2}{l}{\mathsf{distr} \;:\; A \times (B + C) \to (A \times B) + (A \times C)} \\
\multicolumn{2}{l}{\mathsf{distr} = (\mathsf{swap} + \mathsf{swap}) \circ \mathsf{ap} \circ ((\overline{\mathsf{inl}} \triangledown \overline{\mathsf{inr}}) \times \mathsf{id}) \circ \mathsf{swap}}
\end{array}$$

The language also contains a recursion operator: the *hylomorphism* recursion pattern. This was introduced with the first study of recursion patterns in a domain-theoretic setting [13], and was later proved to be powerful enough to allow for the definition of any fixpoint [14]. It is defined as follows.

$$\begin{array}{l}
\mathsf{hylo}_{\mu F} \;:\; (F\ B \to B) \to (A \to F\ A) \to A \to B \\
\mathsf{hylo}_{\mu F} = \lambda g.\ \lambda h.\ \mathsf{fix}(\lambda f.\ g \circ F f \circ h)
\end{array}$$

Function h computes the values passed to the recursive calls, and g combines the results of the recursive calls to compute the final result. The recursion tree of a function defined as a hylomorphism is modeled by μF. The factorial and length functions can then be defined in the point-free language as follows.

$$\begin{array}{l}
\mathsf{fact} \;:\; \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{fact} = \mathsf{hylo}_{\mathsf{List\ Nat}}\ (\mathsf{zero} \triangledown \mathsf{mult})\ ((\mathsf{id} + \mathsf{succ} \vartriangle \mathsf{id}) \circ \mathsf{out}_{\mathsf{Nat}}) \\
\mathsf{length} \;:\; \mathsf{List}\ A \to \mathsf{Nat} \\
\mathsf{length} = \mathsf{hylo}_{\mathsf{Nat}}\ \mathsf{in}_{\mathsf{Nat}}\ ((\mathsf{id} + \mathsf{snd}) \circ \mathsf{out}_{\mathsf{List}\ A})
\end{array}$$

Naturally, other derived operators can be defined using hylomorphism. The following correspond to the well-known *fold* and *unfold* recursion patterns:

$$\begin{array}{ll}
\mathsf{fold}_{\mu F} \;:\; (F\ A \to A) \to \mu F \to A & \mathsf{unfold}_{\mu F} \;:\; (A \to F\ A) \to A \to \mu F \\
\mathsf{fold}_{\mu F} = \lambda g.\ \mathsf{hylo}_{\mu F}\ g\ \mathsf{out}_{\mu F} & \mathsf{unfold}_{\mu F} = \lambda g.\ \mathsf{hylo}_{\mu F}\ \mathsf{in}_{\mu F}\ g
\end{array}$$

3 Point-free Program Transformation

The basic laws of the non-recursive calculus are given in the appendix. We will exemplify their use in the context of a non-trivial program transformation taken from [4]. We resort to the following fold-fusion law to treat recursion:

$$f \circ (\!|g|\!)_F = (\!|h|\!)_F \quad \Leftarrow \quad f \text{ strict} \;\wedge\; f \circ g = h \circ F f \qquad \text{cata-FUSION}$$

where we use the compact notation $(\!|g|\!)_F$ for $\mathsf{fold}_{\mu F}\ g$ (strictness conditions are discussed in detail in [4]). Consider the function `isums::[Int]->[Int]` that computes the initial sums of a list.

```
isums []     = []
isums (x:xs) = map (x+) (0 : isums xs)
```

This can be optimized by introducing an accumulating parameter to store at each point the sum of all previous elements in the list. We first define an operator $\oplus : \mathsf{List\ Int} \times \mathsf{Int} \to \mathsf{List\ Int}$ as $\oplus\ (l, x) = \mathsf{map}_{\mathsf{List}}\ (\overline{\mathsf{plus}}\ x)\ l$. The function isums can then be written as the fold $\mathsf{isums} = (\!|\underline{\mathsf{nil}} \triangledown \oplus \circ \mathsf{swap} \circ (\mathsf{id} \times \mathsf{cons} \circ \underline{\mathsf{zero}} \vartriangle \mathsf{id})|\!)$.

The optimized function isums_t can be calculated from the equation $\mathsf{isums}_t = \overline{\oplus} \circ \mathsf{isums}$, or $\mathsf{isums}_t\ l\ y = \mathsf{map}_{\mathsf{List}}\ (\overline{\mathsf{plus}}\ y)\ (\mathsf{isums}\ l)$ pointwise, which plays the role of specification to the transformation. It can be checked that one obtains by fusion, with F the base functor of lists,

$$\mathsf{isums}_t = (\!|\underline{\underline{\mathsf{nil}}} \triangledown \mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{plus}} \times k)|\!)$$

if there exists a function k such that $\overline{\oplus} \circ \mathsf{cons} \circ \underline{\mathsf{zero}} \vartriangle \mathsf{id} = k \circ \overline{\oplus}$ (the derived constant combinator $\underline{\cdot}$ is defined in the appendix. The following calculation allows to identify $k = \mathsf{cons}^\bullet \circ \mathsf{split} \circ \underline{\mathsf{id}} \vartriangle \mathsf{id}$.

$$\begin{bmatrix}
& \overline{\oplus} \circ \mathsf{cons} \circ \underline{\mathsf{zero}} \vartriangle \mathsf{id} \\
= & \quad \{\, \mathsf{isums\text{-}Aux} \,\} \\
& \mathsf{cons}^\bullet \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \circ \underline{\mathsf{zero}} \vartriangle \mathsf{id} \\
= & \quad \{\, \times\text{-Absor}, \mathsf{zero\ is\ a\ left\text{-}identity\ of\ plus} \,\} \\
& \mathsf{cons}^\bullet \circ \mathsf{split} \circ \underline{\mathsf{id}} \vartriangle \overline{\oplus} \\
= & \quad \{\, \text{const-Fusion} \,\} \\
& \mathsf{cons}^\bullet \circ \mathsf{split} \circ \underline{\mathsf{id}} \circ \overline{\oplus} \vartriangle \overline{\oplus} \\
= & \quad \{\, \times\text{-Fusion} \,\} \\
& \mathsf{cons}^\bullet \circ \mathsf{split} \circ \underline{\mathsf{id}} \vartriangle \mathsf{id} \circ \overline{\oplus}
\end{bmatrix}$$

This uses a new split combinator that internalizes $(\cdot \vartriangle \cdot)$ in the point-free language, as well as an auxiliary law proved elsewhere [4].

$$\begin{aligned} &\mathsf{split}\ :\ (B^A \times C^A) \to (B \times C)^A \\ &\mathsf{split} = \overline{(\mathsf{ap} \times \mathsf{ap}) \circ \pi_1 \times \mathsf{id} \vartriangle \pi_2 \times \mathsf{id}} \end{aligned} \qquad \text{split-Def}$$

$$\overline{\oplus} \circ \mathsf{cons} = \mathsf{cons}^\bullet \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus}) \qquad \text{isums-Aux}$$

Substituting k and converting the resulting definition back to pointwise, one obtains at last the following linear time definition (isums runs in quadratic time).

```
isums_t :: [Int] -> Int -> [Int]
isums_t [] y     = []
isums_t (x:xs) y = (x+y) : isums_t xs (x+y)
```

4 Pointless Haskell: Programming Without Variables

This section describes our implementation of a Haskell library for point-free programming.

Implementing the Basic Combinators. It is well known that the semantics of a real functional programming language like Haskell differs from the standard domain-theoretic characterization, since all data types are by default pointed and lifted (every type has a distinct bottom element). This means that Haskell does not have true categorical products because $(\bot, \bot) \neq \bot$, nor true categorical exponentials because $(\lambda x.\bot) \neq \bot$. For instance, any function defined using pattern-matching, such as `\(_,_) -> 0`, can distinguish between $(\bot, \bot)$ and $\bot$. This problem does not occur with sums because the separated sum also has a distinguished least element.

As discussed in [6], this fact complicates equational reasoning because the standard laws about products and functions no longer hold. In point-free however, as will be shown later, pairs can only be inspected using a standard set of combinators that cannot distinguish both elements, and thus Haskell pairs can safely be used to model products. If we prohibit the use of `seq`, the same applies to functions. Sums are modeled by the standard Haskell data type `Either`.

```
data Either a b = Left a | Right b
```

Concerning the implementation of the terminal object `1`, the special predefined unit data type `()` is not appropriate, because it has two inhabitants `()` and `undefined`. The same applies to any isomorphic data type with a single constructor without parameters. `1` can however be defined as the following data type, whose only inhabitant is `undefined` (to be denoted by `_L`).

```
newtype One = One One
_L = undefined
```

The definition of the point-free combinators in the Pointless library is trivial (see [3] for details). Equipped with these definitions, non-recursive point-free expressions can be directly translated to Haskell. For example, the swap and distr functions can be encoded as follows.

```
swap :: (a,b) -> (b,a)
swap = snd /\ fst
distr :: (c, Either a b) -> Either (c,a) (c,b)
distr = (swap -|- swap) . app . ((curry inl \/ curry inr) >< id) . swap
```

Implementing Functors and Data Types. The implementation of recursive types in Pointless is based on the generic programming library PolyP [15]. This library also views data types as fixed points of functors, but instead of using an explicit fixpoint operator, a non-standard multi-parameter type class with a functional dependency [10] is used to relate a data type `d` with its base functor `f`.

```
class (Functor f) => FunctorOf f d | d -> f
  where inn' :: f d -> d
        out' :: d -> f d
```

The dependency `d -> f` means that different data types can have the same base functor, but each data type can have at most one. The main advantage of

using `FunctorOf` is that predefined Haskell types can be viewed as fixed points of functors (the use of the primes will be clarified later). A relevant subset of PolyP was reimplemented in Pointless according to our own design principles.

To avoid the explicit definition of the map functions, regular functors are described using a fixed set of combinators, according to the definitions

```
newtype Id x        = Id {unId :: x}
newtype Const t x   = Const {unConst :: t}
data (g :+: h) x    = Inl (g x) | Inr (h x)
data (g :*: h) x    = g x :*: h x
newtype (g :@: h) x = Comp {unComp :: g (h x)}
```

The `Functor` instances for these combinators are trivial and omitted here. Given this set of basic functors and functor combinators, the recursive structure of a data type can be captured without declaring new functor data types. For example, the standard Haskell type for lists can be declared as the fixed point

```
instance FunctorOf (Const One :+: (Const a :*: Id)) [a]
    where inn' (Inl (Const _))           = []
          inn' (Inr (Const x :*: Id xs)) = x:xs
          out' []     = Inl (Const _L)
          out' (x:xs) = Inr (Const x :*: Id xs)
```

Naturally, it is still possible to work with data types declared explicitly as fixed points. The fixpoint operator can be defined at the type level using `newtype`.

```
newtype Functor f => Mu f = Mu {unMu :: f (Mu f)}
```

The corresponding instance of `FunctorOf` can be defined once and for all.

```
instance (Functor f) => FunctorOf f (Mu f)
    where inn' = Mu
          out' = unMu
```

The following multi-parameter type class is used to convert values declared using the functor combinators into standard Haskell types and vice-versa.

```
class Rep a b | a -> b
    where to :: a -> b
          from :: b -> a
```

The first parameter should be a type declared using the basic set of functor combinators, and the second is the type that results after evaluating those combinators. The functional dependency imposes a unique result to evaluation. Unfortunately, a functional dependency from `b` to `a` does not exist because, for example, a type A can be the result of evaluating both $\mathsf{Id}\ A$ and $\underline{A}\ B$.

The instances of `Rep` are rather trivial. For the case of products and sums, the types of the arguments should be computed prior to the resulting type. This evaluation order is guaranteed by using class constraints. We give as examples the identity, constant, and product functors:

```
instance Rep (Id a) a
    where to (Id x) = x
          from x = Id x
instance Rep (Const a b) a
    where to (Const x) = x
          from x = Const x
instance (Rep (g a) b, Rep (h a) c) => Rep ((g :*: h) a) (b, c)
    where to (x :*: y) = (to x, to y)
          from (x, y) = from x :*: from y
```

To ensure that context reduction terminates, standard Haskell requires that the context of an instance declaration must be composed of simple type variables. In this example, although that condition is not verified, reduction necessarily terminates because contexts always get smaller. In order to force the compiler to accept these declarations, a non-standard type system extension must be activated with the option `-fallow-undecidable-instances`.

A possible interaction with a Haskell interpreter could now be

```
> to (Id 'a' :*: Const 'b')
('a','b')
> from ('a','b') :: (Id :*: Const Char) Char
Id 'a' :*: Const 'b'
> from ('a','b') :: (Id :*: Id) Char
Id 'a' :*: Id 'b'
```

Note the annotations are compulsory since the same standard Haskell type can represent different functor combinations. This type-checking problem can be avoided by annotating the polytypic functions with the functor to which they should be specialized (similarly to the theoretical notation). Types cannot be passed as arguments to functions, and so this is achieved indirectly through the use of a "dummy" argument. By using the type class `FunctorOf`, together with its functional dependency, it suffices to pass as argument a value of a data type that is the fixed point of the desired functor.

To achieve an implicit coercion mechanism it suffices to insert the conversions in the functions that refer to functors, namely `inn'`, `out'`, and `fmap` (thus the use of primes). The following functions should be used instead.

```
inn :: (FunctorOf f d, Rep (f d) fd) => fd -> d
inn = inn' . from
out :: (FunctorOf f d, Rep (f d) fd) => d -> fd
out = to . out'
pmap :: (FunctorOf f d, Rep (f a) fa, Rep (f b) fb) =>
        d -> (a -> b) -> (fa -> fb)
pmap (_::d) (f::a->b) =
    to . (fmap f :: FunctorOf f d => f a -> f b) . from
```

Implementing Recursion. A polytypic hylomorphism operator can be defined:

```
hylo :: (FunctorOf f d, Rep (f b) fb, Rep (f a) fa) =>
        d -> (fb -> b) -> (a -> fa) -> a -> b
hylo mu g h = g . pmap mu (hylo mu g h) . h
```

Due to the use of implicit coercion it is now possible to program with hylomorphisms in a truly point-free style. For example, the definition of factorial from Section 2 can now be transcribed directly to Haskell. The same applies to derived recursion patterns. Notice the use of bottom as the dummy argument to indicate the type to which a polytypic function should be instantiated.

```
fact :: Int -> Int
fact = hylo (_L :: [Int]) f g   where g = (id -|- succ /\ id) . out
                                      f = one \/ mult
fold   (_::d) g = hylo (_L::d) g out
unfold (_::d) g = hylo (_L::d) inn g
```

5 DrHylo: Deriving Point-free Hylomorphisms

DrHylo is a tool for deriving point-free definitions for a subset of Haskell. The resulting definitions can be executed with the Pointless library. It is based on the well-known equivalence between the simply-typed λ-calculus and cartesian closed categories, first stated by Lambek [12]. One half of this correspondence is testified by a translation from pointwise terms to categorical combinators, later used by Curien to study a new implementation technique for functional languages – the *categorical abstract machine* [5]. We show here how the translation can be extended to handle sums and recursion.

This translation is the starting point for our point-free derivation mechanism. The way variables are eliminated resembles the translation of the lambda calculus into *de Bruijn notation*, where variables are represented by integers that measure the distance to their binding abstractions. Typing contexts are represented by left-nested pairs, as defined by the grammar $\Gamma ::= \star \mid \langle \Gamma, x : A \rangle$, with x a variable and A a type. The translation Φ operates on typing judgments, translated as $\Phi(\Gamma : B \vdash M : A) : B \to A$ according to the rules (typing information omitted)

$$
\begin{array}{ll}
\Phi(\Gamma \vdash \star) & = \mathsf{bang} \\
\Phi(\Gamma \vdash x) & = \mathsf{path}(\Gamma, x) \\
\Phi(\Gamma \vdash MN) & = \mathsf{ap} \circ (\Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N)) \\
\Phi(\Gamma \vdash \lambda x.M) & = \overline{\Phi(\langle \Gamma, x \rangle \vdash M)} \\
\Phi(\Gamma \vdash \langle M, N \rangle) & = \Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N) \\
\Phi(\Gamma \vdash \mathsf{fst}\ M) & = \mathsf{fst} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{snd}\ M) & = \mathsf{snd} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{inl}\ M) & = \mathsf{inl} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{inr}\ M) & = \mathsf{inr} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{case}\ L\ M\ N) & = \mathsf{ap} \circ (\mathsf{either} \circ (\Phi(\Gamma \vdash M) \vartriangle \Phi(\Gamma \vdash N)) \vartriangle \Phi(\Gamma \vdash L)) \\
\Phi(\Gamma \vdash \mathsf{in}\ M) & = \mathsf{in} \circ \Phi(\Gamma \vdash M) \\
\Phi(\Gamma \vdash \mathsf{out}\ M) & = \mathsf{out} \circ \Phi(\Gamma \vdash M) \\
\mathsf{path}(\langle c, y \rangle, x) & = \begin{cases} \mathsf{snd} & \text{if } x = y \\ \mathsf{path}(c, x) \circ \mathsf{fst} & \text{otherwise} \end{cases}
\end{array}
$$

Each variable is replaced by the path to its position in the context tuple, given by function path. The translation of a closed term $M : A \to B$ is a point of

type $1 \to (A \to B)$, which can be converted into the expected function of type $A \to B$ as $\mathsf{ap} \circ (\Phi(\star \vdash M) \circ \mathsf{bang} \vartriangle \mathsf{id})$.

Concerning the translation of the case construct, first notice that $\mathsf{case}\ L\ M\ N$ is equivalent to $(M \triangledown N)\ L$. This equivalence exposes the fact that a case is just an instance of application, and as such its translation exhibits the same top level structure $\mathsf{ap} \circ (\Phi(\Gamma \vdash M \triangledown N) \vartriangle \Phi(\Gamma \vdash L))$. The question remains of how to combine $\Phi(\Gamma \vdash M) : \Gamma \to (A \to C)$ and $\Phi(\Gamma \vdash N) : \Gamma \to (B \to C)$ into a function of type $\Gamma \to (A + B \to C)$. Our solution is based on the internalization of the uncurried version of the either combinator, that can be defined in point-free as follows.

$$\begin{aligned} &\mathsf{either}\ :\ (A \to C) \times (B \to C) \to (A + B) \to C \\ &\mathsf{either} = \overline{(\mathsf{ap} \triangledown \mathsf{ap}) \circ (\mathsf{fst} \times \mathsf{id} + \mathsf{snd} \times \mathsf{id}) \circ \mathsf{distr}} \end{aligned}$$

We give as examples the translations of the swap and coswap functions. The former is translated as the following closed term of functional type, which we then convert to a function of type $A \times B \to B \times A$ and simplify as expected.

$$\Phi(\star \vdash \mathsf{swap}) = \overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} : 1 \to (A \times B \to B \times A)$$

$$\left[\begin{array}{ll} & \mathsf{ap} \circ (\overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} \circ \mathsf{bang} \vartriangle \mathsf{id}) \\ = & \quad \{\ \times\text{-Absor}\ \} \\ & \mathsf{ap} \circ (\overline{\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}} \times \mathsf{id}) \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\ = & \quad \{\ \wedge\text{-Cancel}\ \} \\ & (\mathsf{snd} \circ \mathsf{snd} \vartriangle \mathsf{fst} \circ \mathsf{snd}) \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\ = & \quad \{\ \times\text{-Fusion}\ \} \\ & \mathsf{snd} \circ \mathsf{snd} \circ (\mathsf{bang} \vartriangle \mathsf{id}) \vartriangle \mathsf{fst} \circ \mathsf{snd} \circ (\mathsf{bang} \vartriangle \mathsf{id}) \\ = & \quad \{\ \times\text{-Cancel}\ \} \\ & \mathsf{snd} \vartriangle \mathsf{fst} \end{array}\right.$$

Consider now the translation of the function coswap defined as

$$\begin{aligned} &\mathsf{coswap}\ :\ A + B \to B + A \\ &\mathsf{coswap} = \lambda x.\mathsf{case}\ x\ (\lambda y.\ \mathsf{inr}\ y)\ (\lambda y.\ \mathsf{inl}\ y) \end{aligned}$$

The following result is obtained, which (given some additional facts about either) can be easily simplified into the expected definition $\mathsf{inr} \triangledown \mathsf{inl}$.

$$\overline{\mathsf{ap} \circ (\mathsf{either} \circ (\overline{\mathsf{inr} \circ \mathsf{snd}} \vartriangle \overline{\mathsf{inl} \circ \mathsf{snd}}) \vartriangle \mathsf{snd})} : 1 \to (A + B \to B + A)$$

It can be shown that the translation Φ is sound [5], i.e, all equivalences proved with an equational theory for the λ-calculus can also be proved using the equations that characterize the point-free combinators. Soundness of the translation of sums is proved in [3].

Translating Recursive Definitions. Two methods can be used for translating recursive definitions into hylomorphisms. The first is based on the direct encoding of fix by a hylomorphism, first proposed in [14]. The insight to this result

is that $\mathsf{fix}\ f$ is determined by the infinite application $f\ (f\ (f\ \ldots))$, whose recursion tree is a stream of functions f, subsequently consumed by application. Streams can be defined as $\mathsf{Stream}\ A = \mu(\underline{A} \otimes \mathsf{Id})$ with a single constructor $\mathsf{in} : A \times \mathsf{Stream}\ A \to \mathsf{Stream}\ A$. Given a function f, the hylomorphism builds the recursion tree $\mathsf{in}\ (f, \mathsf{in}\ (f, \mathsf{in}\ (f, \ldots)))$, and then just replaces in by ap. The operator and its straightforward translation are given as follows

$$\begin{array}{l} \mathsf{fix}\ :\ (A \to A) \to A \\ \mathsf{fix} = \mathsf{hylo}_{\mathsf{Stream}\ (A \to A)}\ \mathsf{ap}\ (\mathsf{id} \vartriangle \mathsf{id}) \end{array} \qquad \Phi(\Gamma \vdash \mathsf{fix}\ M) = \mathsf{fix} \circ \Phi(\Gamma \vdash M)$$

Although complete, this translation yields definitions that are difficult to manipulate by calculation. Ideally, one would like the resulting hylomorphisms to be more informative about the original function definition, in the sense that the intermediate data structure should model its recursion tree. An algorithm that derives such hylomorphisms from explicitly recursive definitions has been proposed [9]. In the present context, the idea is to use this algorithm in a stage prior to the point-free translation: first, a pointwise hylomorphism is derived, and then the translation is applied to its parameter functions. DrHylo incorporates this algorithm, adapted to the setting where data types are declared as fixed points, and pattern matching is restricted to sums. Although restrictions are imposed on the syntax of recursive functions, most useful definitions are covered.

Given a single-parameter recursive function defined as a fixpoint, three transformations are produced by the algorithm: one to derive the functor that generates the recursion tree of the hylomorphism ($\mathcal{F}$), a second one to derive the function that is invoked after recursion ($\mathcal{A}$), and a third one for the function that is invoked prior to recursion ($\mathcal{C}$). In general, the function $\mathsf{fix}\ (\lambda f.\ \lambda x.\ L) : A \to B$ is translated as the following hylomorphism.

$$\mathsf{hylo}_{\mu(\mathcal{F}(L))}\ (\lambda x.\ \mathcal{A}(L))\ (\lambda x.\ \mathcal{C}(L)) : A \to B$$

For example, the length function is converted into the following hylomorphism, which can easily be shown to be equal to the expected definition.

$$\begin{array}{l} \mathsf{length}\ :\ \mathsf{List}\ A \to \mathsf{Nat} \\ \mathsf{length} = \mathsf{hylo}_{\mu(\underline{1} \oplus \mathsf{Id})}\ (\lambda x.\ \mathsf{case}\ x\ (\lambda y.\mathsf{in}\ (\mathsf{inl}\ \star))\ (\lambda y.\ \mathsf{in}\ (\mathsf{inr}\ y))) \\ \qquad\qquad\qquad\qquad\quad (\lambda x.\ (\mathsf{out}\ x)\ (\lambda y.\ \mathsf{inl}\ \star)\ (\lambda y.\ \mathsf{inr}\ (\mathsf{snd}\ y))) \end{array}$$

Pattern Matching. In order to apply this translation to realistic Haskell code, we still need to accommodate in our λ-calculus some form of pattern-matching, and data types defined by collections of constructors. It is well-known how to implement an algorithm for defining `FunctorOf` instances for most user-defined data types [15]. This algorithm is incorporated in DrHylo, and since it replaces constructors by their equivalent fixpoint definitions, it suffices to have pattern-matching over the generic constructor in, sums, pairs, and the constant $\star$.

We will now introduce a new construct that implements such a mechanism, but with some limitations: there can be no repeated variables in the patterns, no overlapping, and the patterns must be exhaustive. It matches an expression

against a set of patterns, binds all the variables in the matching pattern, and returns the respective right-hand side.

$$\begin{aligned} P &::= \star \mid x \mid \langle P, P\rangle \mid \mathsf{in}\ P \mid \mathsf{inl}\ P \mid \mathsf{inr}\ P \\ M, N &::= \ldots \mid \mathsf{match}\ M\ \mathsf{with}\ \{P \to N;\ \ldots; P \to N\} \end{aligned}$$

Instead of directly translating this new construct to point-free, a rewriting system is defined that eliminates generalized pattern-matching, and simplifies expressions back into the core λ-calculus previously defined [3]. We remark that since Haskell does not have true products, this rewrite relation can sometimes produce expressions whose semantic behaviour is different from the original.

Consider the Haskell function `\ (x,y) -> 0`. It diverges when applied to `_L`, but returns zero if applied to `(_L,_L)`. This function can be encoded using match and translated into the core λ-calculus using the following rewrite sequence.

$$\begin{aligned} & \lambda z.\mathsf{match}\ z\ \mathsf{with}\ \{\langle x, y\rangle \to \mathsf{in}\ (\mathsf{inl}\ \star)\} \\ \rightsquigarrow\ & \lambda z.\mathsf{match}\ (\mathsf{fst}\ z)\ \mathsf{with}\ \{x \to \mathsf{match}\ (\mathsf{snd}\ z)\ \mathsf{with}\ \{y \to \mathsf{in}\ (\mathsf{inl}\ \star)\}\} \\ \rightsquigarrow\ & \lambda z.\mathsf{match}\ (\mathsf{fst}\ z)\ \mathsf{with}\ \{x \to \mathsf{in}\ (\mathsf{inl}\ \star)\} \\ \rightsquigarrow\ & \lambda z.\mathsf{in}\ (\mathsf{inl}\ \star) \end{aligned}$$

The resulting function is different from the original since it never diverges. Apart from this problem, with this pattern-matching construct it is now possible to translate into point-free many typical Haskell functions, using a syntax closer to that language. For example, distr and the length function can be defined as

$$\begin{aligned} & \mathsf{distr}\ :\ A \times (B + C) \to (A \times B) + (A \times C) \\ & \mathsf{distr} = \lambda x.\mathsf{match}\ x\ \mathsf{with}\ \{\langle y, \mathsf{inl}\ z\rangle \to \mathsf{inl}\ \langle y, z\rangle;\ \langle y, \mathsf{inr}\ z\rangle \to \mathsf{inr}\ \langle y, z\rangle\} \\ & \mathsf{length}\ :\ \mathsf{List}\ A \to \mathsf{Nat} \\ & \mathsf{length} = \mathsf{fix}(\lambda f.\lambda l.\mathsf{match}\ l\ \{\mathsf{in}\ (\mathsf{inl}\ \star) \to \mathsf{in}\ (\mathsf{inl}\ \star);\ \mathsf{in}\ (\mathsf{inr}\ \langle h, t\rangle)\ \to \mathsf{in}\ (\mathsf{inr}\ (f\ t))\}) \end{aligned}$$

6 SimpliFree: Implementing Program Transformations

This section presents SimpliFree, a tool to transform Haskell programs written in the point-free style using Pointless. This tool can be used both to simplify point-free expressions, namely those generated by DrHylo, and to perform some program transformations using fold fusion. For full details on the tool and its implementation the reader is directed to [16].

Basic Principles. SimpliFree is based on the concept of *strategic rewriting*: there is a clear distinction between *rewrite rules*, that just dictate how an equational law should be oriented in order to transform a full term, and *rewriting strategies*, that specify how the basic rules should be applied inside a term and combined in order to produce a full rewrite system.

Likewise to other program transformation tools, such as MAG [7], SimpliFree is based on the notion of *active source*: inside a Pointless program one can also define the rules and strategies that will be used to transform it. When the tool runs with such a program as input, a new Haskell file is produced where:

– Point-free expressions are parsed into an abstract syntax data type `Term`.
– Rewrite rules are converted into functions of type `Term -> m Term`, that try to use Haskell's own pattern matching mechanism to apply a rewrite step to a term (`m` must be a monad belonging to class `MonadPlus`).
– Strategies are built using a basic set of strategy combinators defined in the SimpliFree library, which in turn are defined using the strategic programming library Strafunski [11].

When the resulting file is compiled and executed it returns the transformed Pointless program. Alternatively, it can also be interpreted, allowing the user to inspect the full sequence of rewrite rules applied to a particular expression. Notice that the SimpliFree library already implements some powerful strategies that can be used to effectively simplify most point-free expressions.

Implementing Rules. Rules and strategies are defined in a special annotated block inside the program to be transformed. In particular, rules have a name, and a definition that uses the same concrete syntax of the Pointless library. For example, ×-Cancel, applied to the first argument of a split, and ×-Fusion, applied from right to left, can be defined as follows.

```
{- Rules:
prodCancel1 : fst . (f /\ g) -> f
prodFusionInv : (f . h) /\ (g . h) -> (f/\g) . h
-}
```

One of the fundamental problems to be solved when converting these rules into Haskell functions is how to handle the associativity of composition. In order to avoid implementing matching modulo associativity from scratch, a basic completion procedure had to be implemented on rewrite rules. Sequences of compositions are kept right-associated, and when the left hand side of a rule is a composition, it should be matched not only against a single composition, but also against a *prefix* of a sequence of compositions. For example, the first rule above is translated into the following function.

```
prodCancel1 (FST :.: (f :/\: g)) = return (f)
prodCancel1 (FST :.: ((f :/\: g) :.: x)) = return (f :.: x)
prodCancel1 _ = fail "rule prodCancel1 not applied"
```

Completion is not always this trivial. For example, when a variable is the left argument of a composition there might be the need to try different associations before finding a successful matching. Another problem arises when non-linear patterns are used in the left-hand side of a rule. Since the Haskell matching mechanism cannot handle these patterns, fresh identifiers must be generated to replace repeated variables, and appropriate equality tests have to be introduced in the function bodies. If a rule combines both these problems (such as `prodFusionInv` above) its implementation becomes rather complex.

Strategies. As mentioned above, Strafunski was used in the implementation of strategies and strategy combinators. Strafunski supports two kinds of strategies: *type-preserving* strategies, of type `TP m` for a given monad `m`, that given a term of type `t` return a term of type `m t`; and *type-unifying* strategies, of type `TU a m`, where the result is always of type `m a` regardless of the type of the input. In SimpliFree all strategies are type-unifying. To be more specific they have type `TU Computation m`, where `Computation` is a data type containing both the resulting point-free term, and the list of all intermediate steps in the rewriting sequence. For each step, both the name of the applied rule and the resulting term is recorded.

First of all, there is a basic function that promotes a rule into a strategy:

```
rulePF :: (MonadPlus m) => String -> (Term -> m Term) -> TU Computation m
```

Given a rule, it tries to apply it at most once anywhere inside a term. If successful, it applies an auxiliary type preserving strategy to the full term that associates all compositions to the right. The first argument of `rulePf` is the name of the rule to be recorded.

The library also provides a series of strategy combinators, such as `and`, that given two strategies tries to apply the first and, if successful, applies the second to the result of the first; `or`, that given two strategies tries to apply the first and, if not successful, tries to apply the second; `many`, that repeatedly tries to apply a strategy until it fails; `oneOrMore`, that tries to apply a strategy at least once; and `opt`, that tries to apply a strategy at most once.

Using these strategy combinators we could define the following strategy in a specially annotated block inside a Pointless program.

```
{- Strategies:
simplestrat : compute and (many fold_macros)
compute : simplify and (opt ((oneOrMore unfold_macros) and compute))
simplify : many base_rules

base_rules :  natId1 or natId2 or prodCancel1 or prodCancel2 ...
unfold_macros : exp_unfold or swap_unfold ...
fold_macros : exp_fold or swap_fold ...
-}
```

Each strategy has a name and definition that can refer to rules (defined inside the `Rules` block) or use strategy combinators to build complex rewriting systems. In this example, `simplestrat` tries to apply as many as possible rules from a set of base rules (that encode most of the laws presented in the appendix) in order to simplify a term. When these rules can no longer be applied, it tries to expand one or more macros (such as the definition of common functions like swap, or derived combinators like exponentiation) and returns to the simplification process. If no macros remain to be expanded the simplification stops. In the end it tries to rebuild macros in order to return a more understandable point-free expression to the user. Notice that the translation of strategies to Haskell is trivial: it is only necessary to replace rule invocation by the application of `rulePF` to the respective name.

Example. The SimpliFree tool has a predefined strategy `advstrat` that can be used to effectively simplify the point-free expressions derived by DrHylo. This strategy is an elaboration of the strategy `simplstrat` presented above. In a Pointless program we can specify which of the defined or predefined strategies should be used to transform each point-free declaration. After applying the tool to such a program, the resulting Haskell file contains for each declaration an additional function whose invocation produces the specified transformation, printing at the same time all intermediate steps. The name of this function is just the concatenation of the point-free declaration name and the strategy name (separated by an underscore). For example, after specifying that the swap definition returned by DrHylo should be transformed using the strategy `advstrat`, the following result can be obtained in the Haskell interpreter.

```
*Main> swap_advstrat
app . ((curry ((snd . snd) /\ (fst . snd)) . bang) /\ id)
  = { expCancel }
((snd . snd) /\ (fst . snd)) . (bang /\ id)
  = { prodFusion }
(snd . snd . (bang /\ id)) /\ (fst . snd . (bang /\ id))
  = { prodCancel2 }
(snd . id) /\ (fst . snd . (bang /\ id))
  = { natId2 }
snd /\ (fst . snd . (bang /\ id))
  = { prodCancel2 }
snd /\ (fst . id)
  = { natId2 }
snd /\ fst
```

More elaborate examples, in particular involving the conditional fusion law, can be found in [16].

7 Conclusions and Future Work

We have focused on the most important aspects of each component of the framework; more documentation can be found at the **UMinho Haskell Software** pages:

`http://wiki.di.uminho.pt/wiki/bin/view/PURe/PUReSoftware`

While Pointless has reached a stable stage of development, there are still many points for improvement in the other components. In DrHylo, the translation of recursive functions must be improved with the automatic translation to other standard recursion patterns such as folds, unfolds, and paramorphisms, rather than always resorting to the all-encompassing hylomorphisms.

In SimpliFree, we plan to incorporate other laws for recursive functions, such as unfold-fusion. An immediate goal is to make the fusion mechanism more powerful, to cover at least all the transformations that can be done in state-of-the-art tools such as MAG.

A significant improvement will be the introduction of truly generic laws: in the current version of SimpliFree different fold fusion laws are used for different data types. This is an unfortunate mismatch with the theoretical notation, where recursion patterns and laws are generically defined once and for all.

References

1. John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.
2. Richard Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.
3. Alcino Cunha. *Point-free Program Calculation*. PhD thesis, Departamento de Informática, Universidade do Minho, 2005.
4. Alcino Cunha and Jorge Sousa Pinto. Point-free program transformation. *Fundamenta Informaticae*, 66(4):315–352, 2005. Special Issue on Program Transformation.
5. Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Birkhuser, 2nd edition, 1993.
6. Nils Anders Danielsson and Patrik Jansson. Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction (MPC'04)*, volume 3125 of *LNCS*. Springer-Verlag, 2004.
7. Oege de Moor and Ganesh Sittampalam. Generic program transformation. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Proceedings of the 3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 116–149. Springer-Verlag, 1999.
8. Jeremy Gibbons. Calculating functional programs. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, chapter 5, pages 148–203. Springer-Verlag, 2002.
9. Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82. ACM Press, 1996.
10. Mark Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*, volume 1782 of *LNCS*. Springer-Verlag, 2000.
11. Ralf Laemmel and Joost Visser. Typed combinators for generic traversal. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 137–154, London, UK, 2002. Springer-Verlag.
12. Joachim Lambek. From lambda calculus to cartesian closed categories. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic*, pages 375–402. Academic Press, 1980.
13. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.

14. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM Press, 1995.
15. Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In *Draft proceedings of the 15th International Workshop on the Implementation of Functional Languages (IFL'03)*, 2003.
16. José Proença. Point-free simplification. Technical Report DI-PURe-05.06.01, Universidade do Minho, 2005.

A Laws of the Calculus

$$\pi_1 \vartriangle \pi_2 = \mathsf{id} \qquad \times\text{-Reflex}$$
$$\mathsf{fst} \circ (f \vartriangle g) = f \;\wedge\; \mathsf{snd} \circ (f \vartriangle g) = g \qquad \times\text{-Cancel}$$
$$(f \vartriangle g) \circ h = f \circ h \vartriangle g \circ h \qquad \times\text{-Fusion}$$
$$(f \times g) \circ (h \vartriangle i) = f \circ h \vartriangle g \circ i \qquad \times\text{-Absor}$$
$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-Functor}$$
$$f \vartriangle g = h \vartriangle i \quad \Leftrightarrow \quad f = h \;\wedge\; g = i \qquad \times\text{-Equal}$$
$$f \vartriangle g \text{ strict} \quad \Leftrightarrow \quad f \text{ strict} \;\wedge\; g \text{ strict} \qquad \times\text{-Strict}$$

$$\mathsf{inl} \triangledown \mathsf{inr} = \mathsf{id} \qquad +\text{-Reflex}$$
$$(f \triangledown g) \circ \mathsf{inl} = f \;\wedge\; (f \triangledown g) \circ \mathsf{inr} = g \qquad +\text{-Cancel}$$
$$f \circ (g \triangledown h) = f \circ g \triangledown f \circ h \quad \Leftarrow \quad f \text{ strict} \qquad +\text{-Fusion}$$
$$(f \triangledown g) \circ (h + i) = f \circ h \triangledown g \circ i \qquad +\text{-Absor}$$
$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad +\text{-Functor}$$
$$f \triangledown g = h \triangledown i \quad \Leftrightarrow \quad f = h \;\wedge\; g = i \qquad +\text{-Equal}$$
$$f \triangledown g \text{ strict} \qquad +\text{-Strict}$$

$$\overline{\mathsf{ap}} = \mathsf{id} \qquad \wedge\text{-Reflex}$$
$$f = \mathsf{ap} \circ (\overline{f} \times \mathsf{id}) \qquad \wedge\text{-Cancel}$$
$$\overline{f \circ (g \times \mathsf{id})} = \overline{f} \circ g \qquad \wedge\text{-Fusion}$$
$$f^A \circ \overline{g} = \overline{f \circ g} \qquad \wedge\text{-Absor}$$
$$(f \circ g)^A = f^A \circ g^A \qquad \wedge\text{-Functor}$$
$$\overline{f} = \overline{g} \quad \Leftrightarrow \quad f = g \qquad \wedge\text{-Equal}$$
$$\overline{f} \text{ strict} \quad \Leftrightarrow \quad f \text{ left-strict} \qquad \wedge\text{-Strict}$$

$$\underline{f} = \overline{f \circ \pi_2} \qquad \text{const-Def}$$
$$\underline{f} \circ g = \underline{f} \qquad \text{const-Fusion}$$